OGP Assignment 2015-2016: **The Hillbillies** (Part II)

This text describes the second part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. Therefore, all grades are scored based on this assignment. The assignment is preferably taken in groups consisting of two students; only in exceptional situations the assignment can be worked on individually. In principle, you should compile your solutions for the second and third parts of the assignment with the partner you chose for the first part. You are, however, allowed to start working with a new partner, or to work out the rest of the project on your own. Changes must be reported to ogp-inschrijven@cs.kuleuven.be before the 28th of March 2016. If during the semester conflicts arise within a group, this should be reported to ogp-inschrijven@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

In the course of the assignment, we will create a simple game that is loosely based on *War Craft* and *Dwarf Fortress* – real-time strategy games that involve combat as well as manipulation of the game world. Note that several aspects of the assignment will not correspond to any of the original games. In total, the assignment consists of three parts. The first part focusses on a single class, the second on associations between classes, and the third on inheritance and generics.

The goal of this assignment is to test your understanding of the concepts introduced in the course. For that reason, we provide a graphical user interface for the game and it is up to the teams to implement the requested functionality. This functionality is described at a high level in this document and the student may design and implement one or more classes that provide this functionality, according to their best judgement. Your solution should be implemented in Java 8, satisfy all functional requirements and follow the rules described in this document. The assignment may not answer all possible questions you may have concerning the system itself (functional requirements) or concerning the way it should be worked out (non-functional requirements). You are free to fill in those details in the way that best suits your project. As an example, if the assignment does not impose to use nominal programming, total programming or defensive programming in working out some aspect of the game, you are free to choose the paradigm you prefer for that part. The ultimate goal of the project is to convince us that you master all the underlying concepts of object-oriented programming. The goal is not to hand it the best possible real-time strategy game. Therefore, the grades for this assignment do not depend on correctly implementing functional requirements only. We will pay attention to documentation, accurate specifications, re-usability and adaptability. After handing in your solution to the first part of the assignment, you received feedback on your submission. After handing in the third part of this assignment, the entire solution must be defended in front of Professor Steegmans.

A number of teaching assistants (TAs) will advise the students and answer their questions. More specifically, each team has a number of hours where the members can ask questions to a TA. The TA plays the role of a consultant who can be hired for a limited time. In particular, students may ask the TA to clarify the assignment or the course material, and discuss alternative designs and solutions. However, the TA will not work on the assignment itself. Consultations will generally be held in English. Thus, your project documentation, specifications, and identifiers in the source code should be written in English. Teams may arrange consultation sessions by email to ogp-project@cs.kuleuven.be. Please outline your questions and propose a few possible time slots when signing up for a consultation appointment. To keep track of your development process, and mainly for your own convenience, we encourage you to use a source code management and revision control system such as *Subversion* or *Git*.

1 Assignment

This assignment aims to create a simulation video game that is loosely based on *War Craft* or *Dwarf Fortress*. In THE HILLBILLIES, the player (mostly) indirectly controls a number of hillbilly `Units`. The goal of the game is to maintain the safety of these hillbillies in a hostile three-dimensional game world, destroying hostile `Units` in combat or avoiding them by means of clever manipulation of the world. In the first part of the assignment we focused on a single class `Unit` that implements a basic type of in-game character with the ability to move around, interact with other such characters and manipulate the game world. This second part of the assignment emphasises on associations, adding a number of new classes and interactions to the game. We have indicated new and changed requirements in blue.

Of course, your solution may contain additional helper classes (in particular classes marked *@Value*). In the remainder of this section, we describe the classes `Unit` and `World` in more detail. Unless explicitly stated otherwise, all aspects of your implementation of the class `Unit` shall be specified both formally and informally. Classes other than `Unit` that are specified in this document must be documented informally only. For your support, you will be provided a JAR file containing the user interface for the game together with some helper classes.

1.1 The Class `World`: A Game World

THE HILLBILLIES is played in a cubical game `World` that is composed of X times Y times Z adjointly positioned, non-overlapping *cubes*. The dimensions of a game `World` do not change during the lifetime of this `World`. Each cube is located at a fixed position, denoted by a triple of integer values (x_c, y_c, z_c). The position of the bottom-left-back cube of the game world shall be $(0, 0, 0)$. The position of the top-right-front cube of the game world shall be $(X - 1, Y - 1, Z - 1)$. For the purpose of calculating locations, distances and velocities of game objects, each cube shall be assumed to have a side length $l_c = 1\ m$.

1.1.1 Terrain

A game world shall have geological features, including passable terrain (air, workshop) and impassable (i.e., solid) terrain (rock and wood), that are associated with entire cubes. If a cube of the game world is not assigned a feature explicitly, "air" should be used as the default. Importantly, geological features of the game world may change during the lifetime of the `World`.

Solid terrain cubes must always be attached on (at least) one of their side planes to another solid cube's side plane or to the borders of the `World`. More specifically, a solid cube C at (x_c, y_c, z_c) can be attached to another solid cube at *(a)* $(x_c \pm 1, y_c, z_c)$, *(b)* $(x_c, y_c \pm 1, z_c)$ or *(c)* $(x_c, y_c, z_c \pm 1)$. We call the six cube that match the above specification the *directly adjacent* cubes of C. Groups of directly adjacent cubes may include (at least) one cube that connects to the borders of the game world (i.e., $(0|X-1, ?, ?)$, $(?, 0|Y-1, ?)$, $(?, ?, 0|Z-1)$). Every cube of solid terrain from which there is no path of directly adjacent solid terrain cubes leading to a cube at the borders of the game world, shall cave-in within at most $5\ s$ of game time.

When a solid cube collapses – either as a result of a cave-in or in consequence of a `Unit` manipulating the `World`, the geological property of that cube shall immediately change to "air" and, with a probability of $P = 0.25$, a `Boulder` (rock cube) or a `Log` (wood cube) shall be created at the centre of the collapsed cube.

1.1.2 Game Objects

A `World` may contain an unlimited number of such `Boulders` and `Logs`. A `World` may further contain up to 100 `Units` (cf. Sec. 1.2). Each `Unit` must always belong to a faction. There shall be no more than 5 different active (i.e., non-empty) factions in a `World`. The `World` shall silently reject requests to add new units or factions while these limits are reached.

Game objects such as `Units`, `Boulders` or `Logs` shall be removed from the game world upon death or as they are consumed in certain activities.

1.1.3 Implementation

The game `World` shall provide methods to inspect each individual cube, listing the terrain type and objects currently occupying that cube. The game world shall further provide methods to list all factions, `Units` (individually or per faction), `Boulders` and `Logs` currently present in the game world. A method `advanceTime` must be implemented by the game `World`. This method shall control timed behaviour of the `World` and invoke the `advanceTime` methods of all game objects that are part of the `World`.

1.2 The Class `Unit`

Hillbilly `Units` are considered to be cubical objects that occupy a position (x, y, z) in the game world. Each of them has a name, and a certain weight, strength, agility and toughness. A `Unit` must always belong to a certain faction.

While the game world is using integer coordinates, the position of `Units` shall be treated as double precision floating-point numbers. That is, use Java's primitive type `double` to compute and store these values. Intuitively, a `Unit`'s position (x, y, z) denotes the location of the centre of the `Unit`. Rounding x, y and z down to integer numbers shall yield the position (x_c, y_c, z_c) of a cube of the game world that is said to be occupied by the `Unit`. These concepts are illustrated in Fig. 1. A `Unit` shall never be positioned outside the game world and the components of a `Unit`'s position must at all times be valid numbers. Furthermore, the game `World` cube at the position of the `Unit` must feature a passable (i.e., not solid) terrain type. The class `Unit` shall implement methods to inspect a `Unit`'s position and the position of the game world cube occupied by the unit. Aspects of `Unit` that are concerned with a `Unit`'s position in the game world shall be worked out defensively.

A `Unit`'s name may change during the program's execution. Each name is at least two characters long and must start with an uppercase letter. In the current version, names can only use letters (both uppercase and lowercase),

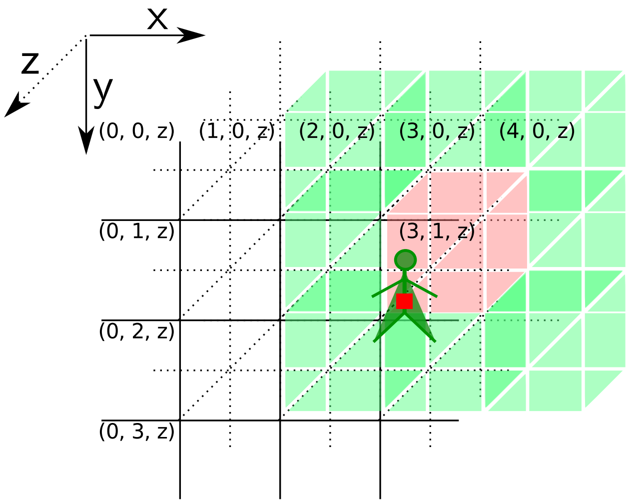

Figure 1: A section of the game world in THE HILLBILLIES and a `Unit`. The given section represents a top-down view on one z-level of the world. The `Unit` sits (or lies) on the edge of cube $3, 1, z$ and we say that it is occupying this cube (shaded in red). The actual position of the `Unit` is indicated by the red square and could, for example, be $3.1, 1.1, z.9$. The `Unit` can move (cf. Sec. 1.2.2) to neighbouring cubes on the same z-level (shaded in green). The `Unit` can further move to neighbouring cubes at $z \pm 1$. These cubes are not highlighted in the figure. They comprise of cubes with the same x and y coordinates as the red and green cubes at directly overlying and underlying z-levels. The boundaries of the game world restrict a `Unit`'s movement options.

quotes (both single and double) and spaces. "James O'Hara" is an example of a well-formed name. It is possible that other characters may be allowed in later versions of the game. However, letters, quotes and spaces will always be legal characters. All aspects related to a `Unit`'s name must be worked out defensively.

A `Unit`'s weight, strength, agility and toughness influence how fast that `Unit` can move, work, and how it behaves in combat. All four attributes may change during the `Unit`'s lifetime and shall be worked out using total programming and integer numbers with values ranging from 1 to 200, inclusively. Importantly, the initial values (i.e., when a new `Unit` object is created) of these attributes shall be in the range of 25 to 100, inclusively, and the weight of a `Unit` must at all times be at least $\frac{strength+agility}{2}$.

Based on their primary attributes, `Units` have a maximum number of hitpoints and a maximum number of stamina points, both being determined as the unit's $200 \cdot \frac{weight}{100} \cdot \frac{toughness}{100}$, rounded up to the next integer. Hitpoints and stamina points can be consumed and regained by means of certain activities. Thus, each `Unit` further has current number of hitpoints and a current number of stamina points which shall always be greater or equal to zero and less or equal to the respective maximum number. Aspects of `Unit` that are concerned with hitpoints and stamina points must be worked out using nominal programming.

A `Unit` shall also have an orientation θ, i.e., a direction the `Unit` is facing. This orientation shall be given as an angle in radians and defaults to $\theta_0 = \frac{\pi}{2}$. The orientation must be updated when the `Unit` is moving or executing other activities (cf. Sec. 1.2.2 ff.). Your implementation of `Unit` must implement a method to inspect the current orientation of a unit. All aspects of `Unit` that concern the orientation attribute shall be worked out using floating-point numbers and total programming.

`Units` can conduct a number of activities, interacting with the game world and other units. In particular, `Units` can move around, work on the cube they are standing on, rest, and attack other units. In the following we will describe these activities in detail. Importantly, the execution of these activities is dependent on *game time*, which is measured in seconds. The class `Unit` shall provide a method `advanceTime` to update the position and activity status of a `Unit`, based on that `Unit`'s current position, attributes and a given duration Δt in seconds of game time. This duration Δt shall never be less than zero and must always be smaller that $0.2\ s$. Unless explicitly specified in the following sections, `advanceTime` and other methods related to the interaction of `Units` with the game world shall be worked out defensively. It is not required to provide formal documentation for the method `advanceTime`.

As `Units` conduct activities, they shall be able to gain experience points, which are to be treated as integer values. For every 10 experience points a `Unit` gains, *one* of that `Unit`'s strength, agility or toughness attributes shall be increased by one.

1.2.1 (Not) Falling

`Units` shall be positioned in the game `World` such that the `Unit` always occupies a passable terrain cube. `Units` further need to stand on (or hang on) solid ground. `Units` are remarkable climbers and are perfectly able to move along a vertical (or even an overhanging) cliff as long as the `Unit` has solid rock or a tree in reach to hold on to. That is, for a `Unit` currently occupying a passable cube with the coordinates (x_c, y_c, z_c), there must be a *neighbouring cube* $(x'_c, y'_c, z'_c) = (x_c \pm 0..1, y_c \pm 0..1, z_c \pm 0..1)$ of impassable (solid) terrain or the `Unit` will fall.

Falling is a special case of basic movement as specified in Sec. 1.2.2: The falling `Unit` shall move with a constant velocity of $\vec{v} = \langle 0, 0, -3 \rangle$ towards the centre of the cube at $(x_c, y_c, z_c - 1)$ until the `Unit` reaches a position that is directly above a solid cube (i.e., for a falling `Unit` currently occupying a non-solid cube at (x_c, y_c, z_c), the cube at $(x_c, y_c, z_c - 1)$ must be solid for the unit to stop falling), or until $z_c = 0$. `Units` cannot start sprinting while falling.

Falling `Units` shall lose 10 hitpoints per Z-level they fall.

1.2.2 Basic Movement

`Units` can move from their current position (x, y, z) to the centre of any neighbouring cube (x', y', z'). As depicted in Fig. 1, for a `Unit` currently occupying a cube with the coordinates (x_c, y_c, z_c), the target cube $(x'_c, y'_c, z'_c) = (x_c \pm 0..1, y_c \pm 0..1, z_c \pm 0..1)$, and $x' = \frac{l_c}{2} + x'_c$, $y' = \frac{l_c}{2} + y'_c$ and $z' = \frac{l_c}{2} + z'_c$, provided that (x', y', z') is within the boundaries of the game world. Here, l_c denotes the length of any side of a cube of the game world, as defined in Sec. 1.1. Importantly, `Units` only move through cubes that feature a passable terrain type and that are neighbouring cubes of solid terrain.

A `Unit`'s movement speed is determined by the relative position of the target cube and the `Unit`'s weight, strength and agility. We compute a `Unit`'s base speed in m/s as $v_b = 1.5 \frac{strengh+agility}{200\frac{weight}{100}}$. For a `Unit` "walking" from some (x, y, z) to a target position (x', y', z') we determine that `Unit`'s walking speed v_w as follows:

$$v_w = \begin{cases} 0.5v_b, & \text{if } z - z' < 0 \\ 1.2v_b, & \text{if } z - z' > 0 \\ v_b, & \text{otherwise} \end{cases}$$

A `Unit` may also choose to do a sprint and move at $v_s = 2v_w$. Sprinting, however, exhausts units quickly, reducing their current stamina by 1 points for each 0.1 second of game time they run. A `Unit` may only start sprinting if that `Unit` is currently moving and the `Unit`'s current stamina is greater than zero. A sprinting `Unit` must stop sprinting as that `Unit`'s stamina reaches zero; the `Unit` may stop sprinting at any time. A `Unit` that stops sprinting before it has reached it's target position (x', y', z') will continue walking towards (x', y', z') with v_w.

The class `Unit` shall provide a method `moveToAdjacent` to initiate movement to a neighbouring cube. Once a unit started moving, subsequent invocations of `advanceTime(`Δt`)` shall lead to updates of that `Unit`'s position. To compute intermediate positions of a `Unit`, we first need to determine that `Unit`'s velocity:

$$d = \sqrt{(x' - x)^2 + (y' - y)^2 + (z' - z)^2}$$

$$\vec{v} = \langle v_x, v_y, v_z \rangle = \langle v_? \frac{(x' - x)}{d}, v_? \frac{(y' - y)}{d}, v_? \frac{(z' - z)}{d} \rangle$$

Here, $v_?$ must be replaced with v_w or v_s, depending on whether the `Unit` is enjoying a quiet stroll in the countryside or hastes towards its destiny. Now the `Unit`'s new position after moving some Δt seconds can be computed as

$$(x, y, z)_{new} = (x, y, z)_{current} + (\vec{v} \cdot \Delta t)$$

$$= ((x_{current} + v_x \Delta t), (y_{current} + v_y \Delta t), (z_{current} + v_z \Delta t))$$

A `Unit` shall stop moving as soon as it reaches or surpasses (x, y, z) within the current time step Δt; the `Unit`'s position shall then be set to (x', y', z'), exactly.

Based on the `Unit`'s velocity, we can also update that `Unit`'s orientation: the orientation attribute of a `Unit` shall be set to $\theta = atan2(v_y, v_x)$, using Java's built-in arctangent function with two arguments.

Movement to a neighbouring cube, unless the `Unit` is currently falling, is only interrupted if the `Unit` is attacked. In that case the `Unit` shall stop moving and execute its defense behaviour (cf. Sec. 1.3.2). The movement of a falling `Unit` cannot be interrupted, neither by fighting nor by any other action. Invocations of `moveToAdjacent` of a `Unit` that is already moving, as well as triggering any other behaviour, shall have no effect. `Unit` shall provide methods to start and stop sprinting and to inspect a `Unit`'s current movement status.

`Unit`s shall gain 1 experience point for every completed movement step. No experience points shall be awarded if the movement was interrupted.

1.3 Extended Movement and Path Finding

`Unit`s typically move to a specific target cube that might be further away than directly neighbouring cubes. Therefore, `Unit` shall provide a method `moveTo` that allows for initiating a complex movement activity that spans multiple `moveToAdjacent`-steps. For a `Unit` that aims to move from some current cube position (x_c, y_c, z_c) to an arbitrary cube (x'_c, y'_c, z'_c), we propose an algorithm in Listing 1. However, students are free to choose and implement other (i.e., faster or more reliable) path finding algorithms such as A^* or Dijkstra's.

Listing 1: Pseudocode of a simple path finding algorithm.

```
let Q be an empty Queue;

function search((position c_0, int n_0))
   let l be a List of cube positions such that each (c ∈ l)
      is neighbouring c_0 ∧
      is of passable terrain ∧
      is neighbouring solid terrain ∧
      (c, n|n ≥ n_0) ∉ Q;
   for each (c ∈ l) Q.append((c, n_0 + 1));

// main loop: path needs to be recomputed to account for
// terrain changes; this can certainly be optimised.
while ((x_c, y_c, z_c) ≠ (x'_c, y'_c, z'_c)) do
   Q.append(((x'_c, y'_c, z'_c), 0));
   while ((x_c, y_c, z_c) ∉ Q ∧ Q.has_next()) do
      (c_0, n_0) = Q.get_next();
      search((c_0, n_0));
   done

   if ((x_c, y_c, z_c) ∈ Q) then
      let (next, n) ∈ Q such that
         next is neighbouring (x_c, y_c, z_c) ∧
         for all (c, m) ∈ Q where c is neighbouring (x_c, y_c, z_c), n ≤ m;
      moveToAdjacent(next);
   else
      terminate pathing;
   fi
done
```

Importantly, pathing to a distant target location can be interrupted by other activities, such as a `Unit`'s need to rest or enemy interaction. In this case, the interrupted `Unit` shall resume pathing to the target position as soon as the interrupting activity is finished. Pathing may be terminated if the target position is not reachable. Invocations of `moveTo` of a `Unit` that is already pathing to some location shall update that `Unit`'s target location with a new target cube.

1.3.1 Work

`Unit` can conduct activities such as digging in the ground, chopping wood, carry objects around, or operating workshops. The class `Unit` shall provide a method `work` to conduct a generic labour at a specified cube position, which must be the unit's current position or a neighbouring cube. The game time needed for finishing a work order depends on the `Unit`'s strength: the `Unit` shall be busy for $\frac{500}{strength}$ s. Floating-point numbers must be used for computing the duration and progress of work activities. Work activities can be interrupted by assigning new tasks to a unit, by fighting or by the unit's requirement to rest. If an activity is interrupted, the interrupted activity shall have no effect on the working `Unit` (other than the game time that passed) or the game `World`.

Depending on the the features an objects present on the target cube, completing a `work` order shall yield results as listed below. Conditions shall be checked in order of appearance, only the first activity for which all conditions are met shall be executed (i.e., `switch-case` semantics).

`Unit` carries a `Boulder` or `Log`: The `Boulder` or `Log` is dropped at the centre of the cube targeted by the labour action.

Workshop cube and one `Boulder` and one `Log` available on cube: The `Unit` will improve their equipment, immediately consuming `Boulder` and `Log`, and increasing the `Unit`'s weight and toughness by one upon completion of the activity.

`Boulder` present: The `Unit` shall pick up the `Boulder`.

`Log` present: The `Unit` shall pick up the `Log`.

Wood cube: The cube collapses as described in Sec. 1.1.1, leaving a `Log`.

Rock cube: The cube collapses as described in Sec. 1.1.1, leaving a `Boulder`.

Importantly, a `Unit` that picks up a `Boulder` or `Log` shall, for the time it carries that `Boulder` or `Log` have an increased weight computed as the `Unit`'s weight plus the weight of the carried object. The temporary weight of a unit may exceed the maximum weight specified earlier.

`Units` shall gain 10 experience points for every completed work order. No experience points shall be awarded for interrupted activities.

1.3.2 Fighting

A `Unit` A can attack other `Units` that occupy the same or a neighbouring cube of the game world. To do so, A and a defending `Unit` D must belong to different factions. D has a chance to either dodge or block the attack, depending on both `Unit`'s strength and agility attributes. If D fails to either dodge or block the attack, D will suffer damage proportional to

A's strength. Conducting an attack shall last 1 s of game time. Defensive actions are instantaneous responses to an attack and require no game time to conduct. The class `Unit` shall implement the specified behaviour in two methods `attack` and `defend`. Units shall gain 20 experience points for every successful attempt at dodging, blocking or attacking. No experience points shall be awarded for unsuccessful attempts.

Dodging. An attacked `Unit` D will always first try and evade the attack by jumping away. The probability for successfully dodging an attack shall be computed as $P_d = 0.20\frac{agility_D}{agility_A}$. If D succeeds in dodging the attack, D shall suffer no damage and shall be moved instantaneously to a random position $(x'_D, y'_D, z'_D) \neq (x_D, y_D, z_D) = (x_D \pm 0..1, y_D \pm 0..1, z_D)$. (x'_D, y'_D, z'_D) must be a valid position featuring passable terrain within the boundaries of the game world. Note that (x'_D, y'_D, z'_D) refers to D's double-precision position. Thus, some $x_D \pm 0..1$ can refer to a new x-position on the same or a neighbouring cube.

Blocking. If an attacked `Unit` D fails to dodge a blow, it will next try to parry the attack. The probability for successfully blocking an attack shall be computed as $P_b = 0.25\frac{strength_D + agility_D}{strength_A + agility_A}$. If D succeeds in blocking the attack, D shall suffer no damage.

Taking Damage. If an attacked `Unit` D fails to dodge or block the attack, D shall suffer damage in terms of a reduction of D's hitpoints by $\frac{strength_A}{10}$.

A's and D's Orientation. Two `Units` A and D fighting each other must also update their orientation θ so that they are facing each other. We can compute and update θ based on the x and y components of the `Unit`'s positions:

$$\theta_A = atan2((y_D - y_A), (x_D - x_A))$$
$$\theta_D = atan2((y_A - y_D), (x_A - x_D))$$

1.3.3 Resting

As `Units` get exhausted or injured, they can rest to recover hitpoints and stamina. A resting unit will recover $\frac{toughness}{200}$ hitpoints or $\frac{toughness}{100}$ stamina points per 0.2 s of game time it spends resting. More specifically, when resting a `Unit` shall always first recover hitpoints until it has reached the `Unit`'s maximum number of hitpoints, and then recover stamina points. If a `Unit` starts resting, it will always rest for at least as long as it takes that `Unit` to recover one hitpoint. This initial recovery period is only interrupted if the

`Unit` is fighting; in that case, neither hitpoints nor stamina are recovered. After the initial period, the `Unit` shall continue resting until it has recovered all hitpoints and stamina points, or until the `Unit` is assigned a new task. `Units` shall automatically rest once every three minutes of game time. The class `Unit` shall implement a method `rest` that initiates resting.

1.3.4 Death

`Units` die if their current number of hitpoints reaches zero. Death units can no longer conduct any activities and all ongoing activities of such a unit shall be interrupted or terminated immediately. If the `Unit` was carrying any objects, these objects shall be dropped at the `Unit`'s current position.

1.3.5 Default Behaviour

`Units` shall have a default behaviour of choosing activities at random. When a `Unit` is not currently conducting an activity, that `Unit` may arbitrarily choose to *(a)* move to a random position within the game world, *(b)* conduct a work task, *(c)* fight potential enemies, or *(d)* rest until it has fully recovered hitpoints and stamina points If a `Unit` is currently moving, it may choose to sprint until it is exhausted.

The default behaviour can be activated and deactivated for each `Unit` individually. To facilitate this, the class `Unit` shall implement two methods `startDefaultBehaviour` and `stopDefaultBehaviour`.

1.4 The Classes `Boulder` and `Log`

`Boulders` and `Logs` are raw materials that are introduced into the game `World` by cave-ins or `Units` manipulating the world. As such, `Boulders` and `Logs` occupy a double-precision position (x, y, z) in the game world. They further have an integer weight between $10 and 50$, inclusively. The weight of a `Boulder` or `Log` shall be assigned at random upon creation of the object; the weight shall never change during the lifetime of the object.

This position must always be located on a passable (non-solid) cube at (x_c, y_c, z_c), for which either $z_c = 0$ or which is located directly above a solid cube (at position $(x_c, y_c, z_c - 1)$). `Boulders` and `Logs` do not actively interact with the game world. They do, however, fall if the above conditions are not met. Once a `Boulder` or `Log` starts falling, it behaves as specified in Sec. 1.2.1 (without losing hitpoints, of course). Thus, the classes `Boulder` and `Log` must implement a method `advanceTime`.

`Boulders` and `Logs` can further be carried around by `Units`. For this purpose, the weight of the `Boulder` or `Log` shall be added to the weight of the carrying `Unit`, affecting that `Unit`'s movement. `Boulder`s or `Log`s shall not be considered as present in the game `World` while they are carried around. Thus, no second `Unit` can attempt to carry a `Boulder` or `Log` that is already being carried. If a `Boulder` or `Log` is dropped it shall inherit the carrying `Unit`'s position and become part of the game `World` again.

2 Reasoning about Floating-point Numbers

Floating-point computations are not exact. This means that the result of such a computation can differ from the one you would mathematically expect. For example, consider the following code snippet:

```
double x = 0.1;
double result = x + x + x;
System.out.println(result == 0.3);
```

The last statement outputs `false`, even though $0.1 + 0.1 + 0.1$ is mathematically equal to 0.3. The output is `false` because the variable `result` holds the value 0.30000000000000004.

A Java `double` consists of 64 bits. Clearly, it is impossible to represent all possible real numbers using only a finite amount of memory. For example, $\sqrt{2}$ cannot be represented exactly and Java represents this number by an approximation. Because numbers cannot be represented exactly, floating-point algorithms make rounding errors. Because of these rounding errors, the expected outcome of an algorithm can differ from the actual outcome.

For the reasons described above, it is generally bad practice to compare the outcome of a floating-point algorithm with the value that is mathematically expected. Instead, one should test whether the actual outcome differs at most ϵ from the expected outcome, for some small value of ϵ. The class `Util` (included in the assignment) provides methods for comparing doubles up to a fixed ϵ.

The course *Numerieke Wiskunde* discusses the issues regarding floating-point algorithms in more detail. For more information on floating-point numbers, we suggest that you follow the tutorial at
http://introcs.cs.princeton.edu/java/91float/.

3 Testing

Write JUnit test suites for the classes `World`, `Unit`, `Boulder` and `Log` that tests each public method. Include these test suites in your submission.

4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on `Unit`. The user interface is included in the assignment as a JAR file. When importing this JAR file you will find a folder `src-provided` that contains the source code of the user interface, the `Util` class and further helper classes. Generally, the files in this folder require no modification from your side. The classes that you develop must be placed in the folders `src` (implementation classes) and `tests` (test classes).

To connect your implementation to the GUI, write a class `Facade` in package `hillbillies.part2.facade` that implements the provided interface `IFacade` from package `hillbillies.part2.facade`. `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, you may execute the `main` method in the class `hillbillies.part2.Part2`. After starting the program, you can press keys to modify the state of the program. Commands are issued by pressing `c` to create a new `Unit`, `Tab` to switch between existing `Units`, and `w`, `r`, `a` to make the currently selected unit work, rest or attack. Movement is controlled by pressing `y`, `u`, `i` to move North (NW, N, NE), `h` and `k` to move West or East, and `b`, `n`, `m` to move South (SW, S, SE). The modifiers `Ctrl` and `Alt` together with the above keys control movement along the z-axis; `Ctrl+j` and `Alt+j` result in moving straight up or down. `Esc` terminates the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Unit`. No additional grades will be awarded for changing the GUI.

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `IFacade`. As described in the documentation of `IFacade`, the methods of your `IFacade` implementation shall only throw `ModelException`. An incomplete test class is included in the assignment to show you what our test cases look like.

5 Submitting

The solution must be submitted via Toledo as a JAR file individually by all team members before the 10th of April 2016 at 11:59 PM. You can generate a JAR file on the command line or using eclipse (via `export`). Include all source files (including tests) and the generated class files. Include your name, your course of studies and a link to your code repository in the comments of your solution. When submitting via Toledo, make sure to press `OK` to confirm the submission!